Analysis and Comparison of Three Algorithms for the Vertex Cover Problem on Large Graphs with Low Memory Capacities

Eric Angel, Romain Campigotto, Christian Laforest

Analysis and Comparison of Three Algorithms for the Vertex Cover Problem on Large Graphs with Low Memory Capacities*

Eric Angel
Lab. IBISC, Université d'Evry, France
eric.angel@ibisc.fr

Romain Campigotto
Lab. IBISC, Université d'Evry, France
romain.campigotto@ibisc.fr

Christian Laforest
Lab. LIMOS, CNRS – Université B. Pascal (Clermont 2), France
christian.laforest@isima.fr

Abstract

In this paper, we consider the classical **NP**-complete VERTEX COVER problem in *large graphs*. We assume that the size and the access to the input graph impose the following constraints: (1) the input graph must not be modified (*integrity* of the input instance), (2) the computer running the algorithm has a memory of limited size (compared to the graph) and (3) the result must be sent to an output memory once a new piece of solution is calculated. Despite the severe constraints of the model, we propose three algorithms that satisfy them. We derive exact formulas giving the *expected size* of the solution they return. This allows us to compare them, in an analytic way. Then, we consider their complexities. We give exact formulas expressing the *expected number of requests* they perform on the input graph. Again, we compare them analytically. For both comparisons, we show that none of them is better than the two others.

The formulas we give can help users to estimate the best balance between quality of the solution and performance.

Key words: *large graphs, vertex cover, mean analysis of algorithms*

1 Introduction

Most of the known optimization algorithms need to explore, mark, modify, etc. the instance given as input before producing their results. To do that, the instance is entirely loaded into the memory of the computer and is manipulated by the algorithm. Often, "extra" data structures are also necessary to memorize parameters useful all along the computation or to update the current solution that will be returned as the final product of the program.

However, this classical model is no more adapted for many new computing applications. Indeed, nowadays, many fields such as biology, meteorology, finance, etc. produce very large amount of

*Work partially supported by the projects *ToDo* (French ANR) and *Approximation rapide* of GDR-RO.

data. These data are usually stored on large databases, called *data warehouses*, in order to be exploited and analyzed. These data are collected by a *source* that can be a laboratory (collection of experimental results or physical measures) or a company (collection of financial values for example). This source can open the access of its collected data to external *partners*[1]. However, as the data often result from heavy and/or costly experimental process, they must not be corrupted by the manipulations of the partners. This means that the data must be read-only and must be preserved from modifications.

However, a partner does not always have a machine with the capacity to load the whole data and as the treatment of such huge data takes time and it cannot in general allocate all its computers during such a long period. For simplicity here, we suppose that it allocates only one computer with standard memory capacities.

Our General Model of Access to Data. With the previous discussions, we model the situation as follows (we give an illustration in Fig. 1). We assume there is one standard computer, called the "Processing Unit", for accessing data and running algorithms. The input data are stored on a data warehouse called "Input data". As the solution of the computation can be large, we suppose that it is stored on an external memory (e.g. a hard disk or a data warehouse) called "Result". We enumerate now the main constraints of our general model.

C_1. The input data cannot be modified; the *integrity* of input data must be preserved.

C_2. The processing unit has a "small" memory space (compared to the huge size of the data/instance).

C_3. The solution must be sent piece by piece to an external memory, called here "Result", as soon as it is produced.

Constraint C_2 implies that the instance cannot be loaded into the memory of the processing unit (see hypothesis above). The constraint C_3 comes from the fact that in many cases, the solution is (in order of magnitude) as large as the input data, i.e. impossible to be stored in the memory of the processing unit. Because of memory constraints, the solution cannot fit in memory of the processing unit. Hence, using intermediate solutions to construct the final one can here be complex, take time and memory since this imply to reload the appropriated part of the current solution from the result machine. To avoid such complex mecanisms, we adopt here a radical point of view in proposing methods that scan data and send final results as soon as they are produced, without keeping in memory trace of past computation and without modifying past part of the solution.

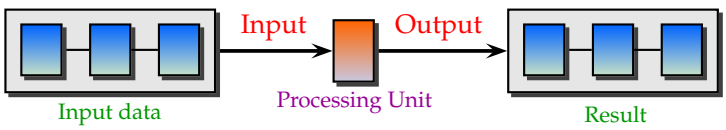

Figure 1: Overview of the model

[1]We do not treat at all here problems related to rights of access to these data. We suppose that the partners have all the appropriated rights to *read* the data.

The Vertex Cover Problem. We have chosen to study in this paper the well-known VERTEX COVER problem, a classical **NP**-complete optimization graph problem [5], that has received a particular attention for the last few decades. In particular, this problem occurs in many concrete applications, such as the network monitoring [12, 17] or the resolution of biological conflicts [12, 15], and many approximation algorithms have been proposed (see for example the section of [3] devoted to this problem).

Notations. Graphs $G = (V, E)$ considered throughout this paper are undirected, simple, unweighted and represent the *instance* to be treated here. We denote by n the number of vertices ($n = |V|$) and by m the number of edges ($m = |E|$). For any vertex $u \in V$, we denote by $N(u)$ the set of *neighbors* of u (i.e. the set of vertices sharing an edge with u), $d(u) = |N(u)|$ the *degree* of u (i.e. the number of neighbors) and Δ the *maximum degree* of vertices of G.

Definition of the Vertex Cover Problem. A *cover* C of G is a subset of vertices such that every edge contains (or *is covered by*) at least one vertex of C, that is $C \subseteq V$ and $\forall e = uv \in E$, one has $u \in C$ or $v \in C$ (or both). The VERTEX COVER problem is to find a cover of minimum size.

Example of Application on the Vertex Cover Problem in Our Model. Let us consider the Single Nucleotide Polymorphism (SNP[2], pronounced "snip") Haplotype Assembly Problem [8]. In this problem, geneticists are interested to the genetic differences among individuals. More precisely, they want to determine haplotypes for large numbers of individuals, i.e. sets of variants genetically linked because of their proximity on the genome.

Let $\mathcal{G} = (\mathcal{S}, \mathcal{C})$ be a *SNP conflict graph*, constructed from DNA sequences, SNPs and experimental values. In this graph, each vertex $s_i \in \mathcal{S}$ represents a SNP and each edge $\{s_i, s_j\} \in \mathcal{C}$ represents a conflict between two distinct SNPs s_i and s_j (for more details about this notion, see [14]). The SNP Assembly Problem is to maximize the number of SNPs which are not in conflict. In other words, the goal is to remove the smallest subset $\mathcal{S}'$ of $\mathcal{S}$ from $\mathcal{G}$, such that the induced subgraph $\mathcal{G} \setminus \mathcal{S}'$ contains no edge; that is to find a cover of minimum size in $\mathcal{G}$.

From massive experimental measures, one can generate very large DNA sequences and very large number of SNPs (in a DNA Sequencing Center for example) and then easily create a (very large) *SNP conflict graph* stored on a data warehouse. These data/graphs can be shared, via read-only access, with scientists for various computational experiments, measures, etc.

A geneticist, who wants to resolve biological conflicts in such a particular graph, does not necessarily have powerful computers to make the work. Therefore, he has limited possibilities, e.g. he cannot copy the whole graph into the memory of its computer (but he can let a software run for several days). Thus, a simple process of computation must be implemented on its machine, getting the *SNP conflict graph* piece by piece by sending requests to the data warehouse, perform processing on each of these pieces and send the result to an external local hard disk for example.

In this paper, we propose and compare algorithms that have all the features to run under such particular constraints and low powerful environments.

[2] A *SNP* is a single base mutation in DNA.

Quick Overview of Existing Algorithms for Vertex Covering. Many algorithms have been proposed for the VERTEX COVER problem. As it is **NP**-hard, most of the methods are approximation algorithms or heuristics. Here, we give a rapid overview of these methods.

A well-known heuristic is to select a vertex of maximum degree and delete this vertex and its incident edges from the graph, until all edges have been removed [10]. It has an approximation ratio in $\mathcal{O}(\log \Delta)$. Another popular algorithm, with the best known constant approximation ratio, 2, is to construct a maximal matching of the input graph and return the vertices of the matching (see [3]). To compute such a solution, an edge is randomly chosen, and its two endpoints with their incident edges are deleted from the graph, until all edges have been removed. For these algorithms, in order to delete a vertex and its incident edges, we have to modify the input graph or store information on deleted elements into the memory of the processing unit, and that does not satisfy constraints C_1 and C_2. Another well-known algorithm is to construct a *DFS* spanning tree and select its internal nodes [13]. It has an approximation ratio of 2. During the computation of a *DFS* spanning tree, we have to keep several vertices into memory (those which are being explored and those which have been explored) or to mark these vertices in the input graph, and again that does not satisfy our constraints.

The best known algorithm has an approximation ratio of $2-\Theta\left(\frac{1}{\sqrt{\log n}}\right)$. It is based on semidefinite programming relaxation (see [7]). This kind of method requires to fit entirely the graph into memory, that does not satisfy C_2.

Thus, there are many algorithms for the VERTEX COVER problem but there does not seem to be a way to implement them in order to satisfy the constraints C_1, C_2 and C_3 given in the introduction.

Organization of the Paper. Despite the very severe constraints of the model and the intrinsic difficulty of the VERTEX COVER problem (**NP**-complete), we describe in Sect. 2 three algorithms adapted to our model.

To compare them, we propose in Sect. 3 general analytical formulas giving the exact expected size of the vertex cover produced. This leads us to show that none of them is better than the two others.

To go further in the comparison, we give in Sect. 4 general formulas giving the maximum number and the expected number of requests made by the algorithms on the "Input data" warehouse. This is a measure of complexity of our algorithms. We show that based on this measure, none of the algorithm is better than the two others.

We conclude and give perspectives in Sect. 5.

2 The Algorithms $\mathcal{LL}$, $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$

We describe in this section three algorithms suitable to our model: $\mathcal{LL}$ (*ListLeft*), $\mathcal{SLL}$ (*Sorted-LL*) and $\overleftarrow{\mathcal{SLL}}$ (*Anti Sorted-LL*).

Labeling of Nodes, Left and Right Neighbors. In real applications, the vertices have labels (depending on the applications domain) which are assumed to be pairwise distinct and can

be ordered (e.g. by lexicographic order). We formalize this as follows. In a *labeled graph*, denoted by $G = (V, L, E)$, the vertices of G are labeled by a given function L such that for each vertex $u \in V$, a unique *label* $L(u) \in \{1, \ldots, n\}$. We denote by $\mathbb{L}(G)$ the set of all possible labelings L for a graph $G = (V, E)$. Given a labeled graph $G = (V, L, E)$ and a vertex $u \in V$, v is called a *right neighbor* (resp. *left neighbor*) of u if $v \in N(u)$ and if v has a label larger (resp. smaller) than u.

Description of the Algorithms. We give now a basic description of our algorithms, based on the previous notions. We give later the way they can be implemented in the model of Fig. 1 to satisfy constraints C_1, C_2, C_3.

Algorithm 1 ($\mathcal{LL}$)**.** Let $G = (V, L, E)$ be a labeled graph. For each vertex $u \in V$, u is added to the cover if it has at least one *right neighbor*.

Algorithm 2 ($\mathcal{SLL}$)**.** Let $G = (V, L, E)$ be a labeled graph. For each vertex $u \in V$, u is added to the cover if $\exists v \in N(u)$ such that $d(v) < d(u)$ or if u has at least one *right neighbor* with the same degree.

Algorithm 3 ($\overleftarrow{\mathcal{SLL}}$)**.** Let $G = (V, L, E)$ be a labeled graph. For each vertex $u \in V$, u is added to the cover if $\exists v \in N(u)$ such that $d(v) > d(u)$ or if u has at least one *left neighbor* with the same degree.

Approximation Ratios. It can be easily seen that these algorithms always return a vertex cover of the input graph. $\mathcal{LL}$ and $\overleftarrow{\mathcal{SLL}}$ have an approximation ratio of at least Δ. Indeed, on stars $\mathcal{LL}$ can return all the leaves, if the center is labeled by n, and $\overleftarrow{\mathcal{SLL}}$ returns all the leaves. It has been proved in [4] that $\mathcal{SLL}$ (presented as a *list algorithm*) has an approximation ratio of at most $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$.

Details on the Model and Satisfaction of the Constraints C_1, C_2 and C_3. We suppose that the data warehouse stores the labeled graph $G = (V, L, E)$ in the form of an adjacency list in which vertices and their neighbors are stored in an arbitrary order (not necessarily following the labels).

If the degrees of the vertices are not stored in the input data unit, only $\mathcal{LL}$ can be used. If we suppose that, in addition, the degrees are stored in a table (with direct access in the input unit), $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$ can also be executed. The table of degrees must have been stored and computed when the graph has been constructed; we suppose here that it is available.

The processing unit (running $\mathcal{LL}$, $\mathcal{SLL}$ or $\overleftarrow{\mathcal{SLL}}$) sends *requests* to the data warehouse to scan G vertex by vertex and for each current vertex u (its label and its degree if needed), scans its neighbors (their labels and their degrees if needed) one by one. When the processing unit decides that a vertex u belongs to the solution (applying the conditions given in the descriptions of the algorithms above), u is put immediately and definitively into the cover (it is sent to "Result"). Then, the processing unit asks for the next vertex (and its neighbors) from the data warehouse; otherwise, it must scan all the neighbors of u (and, at the end, require the next vertex like in the previous case). We suppose that the "Input data" warehouse has the ability to do all these

operations in an efficient way (returning the labels and the degrees, going to the next neighbor, the next vertex, etc.).

In this model, the three algorithms satisfy the constraints C_1 (the instance is not loaded), C_2 (at any moment the processing unit only has two labels in memory and two degrees) and C_3 (the current piece of the solution is sent as soon as it is produced).

It is worth to notice that $\mathcal{LL}$ can be adapted to the streaming model (see [9] for a survey), since it requires only labels of vertices to compare them.

3 Mean Analysis about Quality of Solutions

These three algorithms work deterministically on any given labeled graph. However, the labels of vertices are often totally arbitrary and only come from the application domains. Different labelings can give different results, i.e. covers of different sizes. In this section, we compare these algorithms with respect to the size of the vertex cover they return. Since there are $n!$ possible labelings in $\mathbb{L}(G)$, we assume that each one can occur with a probability $\frac{1}{n!}$.

We give in Theorem 3.1 exact formulas corresponding to the expected size of solution constructed by $\mathcal{LL}$, $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$ on any graph G. For that, we introduce additional notations.
Let $S = V \setminus \{u \mid \exists v \in N(u), d(v) < d(u)\}$ (resp. $\overleftarrow{S} = V \setminus \{u \mid \exists v \in N(u), d(v) > d(u)\}$) be the set of vertices with no neighbor of lower (resp. greater) degree. Let $\sigma(u) = |\{v \mid v \in N(u) \wedge d(v) = d(u)\}|$ be the number of neighbors of u having the same degree as that of u.

Theorem 3.1. *Let $G = (V, E)$ be any graph. Let $\mathbb{E}[\mathcal{A}(G)]$ be the expected size of the solution constructed by algorithm $\mathcal{A}$ on G. By considering all the labelings of $\mathbb{L}(G)$ with equiprobability assumption, we have*

$$\mathbb{E}[\mathcal{LL}(G)] = n - \sum_{u \in V} \frac{1}{d(u)+1} \ , \tag{1}$$

$$\mathbb{E}[\mathcal{SLL}(G)] = n - \sum_{u \in S} \frac{1}{\sigma(u)+1} \ , \tag{2}$$

$$\mathbb{E}[\overleftarrow{\mathcal{SLL}}(G)] = n - \sum_{u \in \overleftarrow{S}} \frac{1}{\sigma(u)+1} \ . \tag{3}$$

Proof. We give the proof for $\mathcal{LL}$ and for $\mathcal{SLL}$. The proof for $\overleftarrow{\mathcal{SLL}}$ is similar to the $\mathcal{SLL}$ one.

Proof for $\mathcal{LL}$. Let $G = (V, L, E)$ be any labeled graph. Let $C_{\mathcal{LL}}$ be a cover constructed by $\mathcal{LL}$ on the labeled graph G. Let us consider a vertex u of G. u is not selected by $\mathcal{LL}$ if and only if it has no right neighbor, which means that all its neighbors have labels smaller than it. Since we consider a uniform distribution over the set of $n!$ possible labelings, this event appears with a probability of $\frac{d(u)!}{(d(u)+1)!}$. Indeed, if we sort u and the $d(u)$ vertices of $N(u)$ by increasing order of labels, there are $(d(u)+1)!$ possible permutations, and the number of permutations such that u is in the last position is $d(u)!$. Thus, $\mathbb{P}[u \in C_{\mathcal{LL}}] = 1 - \frac{1}{d(u)+1}$ and the result follows by summing those probabilities for each vertex u of G.

Proof for $\mathcal{SLL}$. Let $G = (V, L, E)$ be any labeled graph. Let $C_{\mathcal{SLL}}$ be a cover constructed by $\mathcal{SLL}$ on the labeled graph G. Let us consider a vertex u of G. If $u \notin S$, it means that there exists a vertex $v \in N(u)$ such that $d(v) < d(u)$. So, for all the vertices of $V \setminus S$, we have $\mathbb{P}[u \in C_{\mathcal{SLL}} \mid u \notin S] = 1$. Also, if $u \in S$, then it is selected by $\mathcal{SLL}$ if it has at least one right neighbor with the same degree. By following the same principle as for $\mathcal{LL}$, for all the vertices of $V \setminus S$, we have $\mathbb{P}[u \in C_{\mathcal{SLL}} \mid u \in S] = 1 - \frac{1}{\sigma(u)+1}$. The result follows by summing those probabilities for each vertex u of G. □

One can note similarities between the proof of Theorem 3.1 for $\mathcal{LL}$ and a result of *Caro* and *Wei* on the size of an *Independent Set* in a graph (see [1]).

Theorem 3.2. *Among $\mathcal{LL}$, $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$, no algorithm can be elected as the best one: there exist graphs for which each algorithm returns, in expectation, a cover smaller than the two others.*

Proof. We show that, for each algorithm, there exist graphs for which it is the best in expectation.

- Let S_n be a star with n vertices. If we apply resp. (1), (2) and (3) on S_n, for all $n > 2$, we have
$$\mathbb{E}[\mathcal{LL}(S_n)] \;=\; n - \frac{n-1}{2} - \frac{1}{n} \;=\; \frac{n}{2} - \frac{1}{n} + \frac{1}{2} \;.$$
For $\mathcal{SLL}$, the set S contains all the leaves of S_n. Thus, we have
$$\mathbb{E}[\mathcal{SLL}(S_n)] \;=\; n - n + 1 \;=\; 1 \;.$$
For $\overleftarrow{\mathcal{SLL}}$, the set $\overleftarrow{S}$ only contains the center of S_n. Hence, we have
$$\mathbb{E}[\overleftarrow{\mathcal{SLL}}(S_n)] \;=\; n - 1 \;.$$
We can easily see that $\mathbb{E}[\mathcal{SLL}(S_n)] \;<\; \mathbb{E}[\mathcal{LL}(S_n)] \;<\; \mathbb{E}[\overleftarrow{\mathcal{SLL}}(S_n)]$. Note that $\mathcal{SLL}$ is optimal for S_n.

- Let $GR_{p\times q}$ be a grid graph with $n = p \times q$ vertices. $\forall p, q > 2$, we have
$$\mathbb{E}[\mathcal{LL}(GR_{p\times q})] \;=\; n - \frac{(p-2)(q-2)}{5} - \frac{2(p+q-4)}{4} - \frac{4}{3} \;=\; \frac{4n}{5} - \frac{p+q}{10} - \frac{2}{15} \;.$$
For $\mathcal{SLL}$, the set S contains all the vertices which are neighbors to the border and the corner vertices of $GR_{p\times q}$. So, we have
$$\mathbb{E}[\mathcal{SLL}(GR_{p\times q})] \;=\; n - \frac{(p-4)(q-4)}{5} - \frac{2(p+q-8)}{3} - 4 \;=\; \frac{4n}{5} + \frac{2(p+q)}{15} - \frac{28}{15} \;.$$
For $\overleftarrow{\mathcal{SLL}}$, the set $\overleftarrow{S}$ contains all the border and corner vertices of $GR_{p\times q}$. Therefore, we have
$$\mathbb{E}[\overleftarrow{\mathcal{SLL}}(GR_{p\times q})] \;=\; n - \frac{(p-4)(q-4)}{5} - \frac{2(p+q-8)}{4} - \frac{4}{3} \;=\; \frac{4n}{5} + \frac{3(p+q)}{10} - \frac{8}{15} \;.$$
Thus, we can see that $\mathcal{LL}$ is better in expectation than $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$ on grid graphs.

- Let AI_a^+ be a special bipartite graph with $n = 2a^2 + a - 1$ vertices. In AI_a^+, the set of vertices is $X_1 \cup X_2 \cup Y_1 \cup Y_2$, with $X_1 = \{v_1, \ldots, v_{a^2-2}\}$, $Y_1 = \{w_1, \ldots, w_{a^2}\}$, $X_2 = \{z_1, \ldots, z_a\}$ and $Y_2 = \{t\}$. The set of edges is $v_i w_j \ \forall i, j$, $z_i w_{a(i-1)+k}$ for $k = 1, \ldots, a$ and $i = 1, \ldots, a$; and $t z_i \ \forall i$. An example is given in Fig. 2. Note that an AI_a^+ graph is an extension of graphs presented in [4].

  We consider that $a > 2$. The set of vertices $V = X_1 \cup Y_1 \cup X_2 \cup Y_2$ is constituted as follows: X_1 contains $a^2 - 2$ vertices of degree a^2, Y_1 contains a^2 vertices of degree $a^2 - 1$, X_2 contains a vertices of degree $a + 1$, and Y_2 contains 1 vertex of degree a. Thus, for $\mathcal{LL}$, we have

$$\begin{aligned} \mathbb{E}[\mathcal{LL}(AI_a^+)] &= n - \frac{a^2-2}{a^2+1} - \frac{a^2}{a^2-1+1} - \frac{a}{a+2} - \frac{1}{a+1} \\ &= n - 1 - \frac{a^2-2}{a^2+1} - \frac{a}{a+2} - \frac{1}{a+1} \ . \end{aligned}$$

  For $\mathcal{SLL}$, the set S only contains the vertex t of Y_2. Thus, we have

$$\mathbb{E}[\mathcal{SLL}(AI_a^+)] = n - 1 \ .$$

  For $\overleftarrow{\mathcal{SLL}}$, the set $\overleftarrow{S}$ contains the $a^2 - 2$ vertices of X_1. Therefore, we have

$$\mathbb{E}[\overleftarrow{\mathcal{SLL}}(AI_a^+)] = n - a^2 + 2 \ .$$

  We can see that $\mathcal{LL}$ is better than $\mathcal{SLL}$. We compare $\overleftarrow{\mathcal{SLL}}$ with $\mathcal{LL}$:

$$\mathbb{E}[\mathcal{LL}(AI_a^+)] - \mathbb{E}[\overleftarrow{\mathcal{SLL}}(AI_a^+)] = a^2 - 3 - \frac{a^2-2}{a^2+1} - \frac{a}{a+2} - \frac{1}{a+1} > 0$$

  when $a > 3$, because $\frac{a^2-2}{a^2+1} < 1$, $\frac{a}{a+2} < 1$ and $\frac{1}{a+1} < 1$, that implies $\frac{a^2-2}{a^2+1} + \frac{a}{a+2} + \frac{1}{a+1} < 3$. Hence, $\mathbb{E}[\overleftarrow{\mathcal{SLL}}(AI_a^+)] < \mathbb{E}[\mathcal{LL}(AI_a^+)] < \mathbb{E}[\mathcal{SLL}(AI_a^+)]$. Note that $\mathcal{SLL}$ always returns a worst solution (of size $n - 1$) on any AI_a^+ graph.

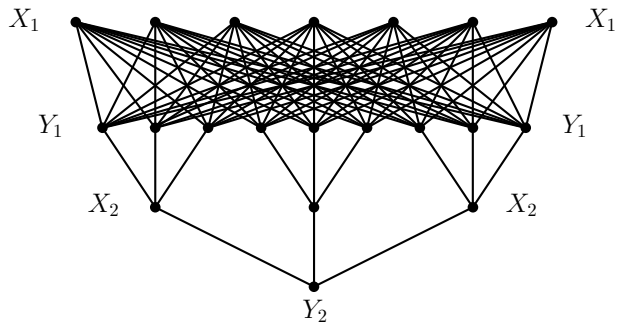

Figure 2: Example of AI_a^+ graph with $a = 3$

□

Applications of (1), (2) and (3) on another classes of graphs can be found in [2].

Special Properties of $\mathcal{LL}$. We show here that for any graph G, $\mathcal{LL}$ can construct an optimal cover for any graph G in the best case or a very large cover in the worst case.

Lemma 3.1. *For any graph G, there exists a labeling function $L^* \in \mathbb{L}(G)$ such that $\mathcal{LL}$ returns an optimal solution on the labeled graph $G = (V, L^*, E)$.*

Proof. Let C^* be an optimal cover. It is easy to show that $V \setminus C^*$ is an independent set and that each $u \in C^*$ has at least a neighbor in $V \setminus C^*$ (otherwise, u and all its neighbors would be in C^*, thus C^* would not be optimal). The labeling function L^* we propose is one such that vertices of C^* get labels between 1 and $|C^*|$ and vertices of $V \setminus C^*$ get labels between $|C^*|+1$ and n. If algorithm $\mathcal{LL}$ is executed on such a labeled graph, it returns all the vertices of C^* (since each vertex u of C^* has at least a neighbor in $V \setminus C^*$ with a higher label) and no vertex of $V \setminus C^*$ (because $V \setminus C^*$ is an independent set and thus each vertex in this set only has neighbors in C^*, i.e. "on its left"). □

Lemma 3.2. *For any graph G, there exists a labeling function $L_w \in \mathbb{L}(G)$ such that $\mathcal{LL}$ returns a cover of size $n - c$ on the labeled graph $G = (V, L_w, E)$, with c the number of connected components of G ($c = 1$ if G is connected). This bound is tight: $\mathcal{LL}$ cannot construct a cover of size more than $n - c$.*

Proof. First, we consider a graph G with $c = 1$ connected component. Let T be any spanning tree of G. Let r be any vertex of T. The labeling function $L_w \in \mathbb{L}(G)$ labels the vertices as follows. Vertex r gets label n. The d_1 neighbors/children of r in T get the d_1 labels $(n - d_1, \ldots, n - 1)$; the d_2 vertices at distance 2 get the d_2 preceding labels $(n - d_1 - d_2, \ldots, n - d_1 - 1)$, etc. until each vertex receives a label, level by level (see Fig. 3 for an illustration). With this labeling, since T is a spanning tree, each vertex $u \neq r$ has at least one right neighbor: its parent v in the tree T rooted in r. Hence, the execution of $\mathcal{LL}$ on this labeled graph $G = (V, L_w, E)$ will return all the vertices, except the root r, which is the vertex labeled with the maximum value. This is the maximum size achievable, since $\mathcal{LL}$ never put in a cover the vertex with the larger label (since it cannot have a right neighbor).

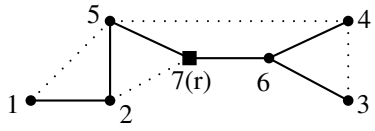

Figure 3: Example of a labeled spanning tree of a graph. *Dotted lines* correspond to edges which are present in the graph but not in the spanning tree.

If G is not connected, we can apply the previous labeling and analysis on each connected component of G. □

4 Analysis of the Number of Requests

In Sect. 2, we have seen that, during the execution of algorithms, the processing unit gets vertices one by one, in any order of labels (not necessarily from 1 to n). Moreover, the neighbors of a vertex u are also obtained one by one, in any order. That implies two situations.

1. If u is not sent to the cover by examining the current neighbor, the processing unit retrieves a neighbor of u which has not yet been scanned. If there is no remaining neighbor, i.e. when u has been compared with all of its neighbors, it decides definitively that u is not in the cover.

2. If u is sent to the cover because of the examination of the current neighbor, the system doesn't need to go further and to compare u with its remaining neighbors. Hence, the processing unit is not required to retrieve all the neighbors of a vertex.

We call *request* the action of getting a neighbor (its label, and its degree, if needed) which has not yet been scanned. In this section, we evaluate the *number of requests* made by the three algorithms to construct their solutions. Given a labeled graph, this number depends on the order in which neighbors of vertices are sent to the algorithm.

In our model, the processing unit takes longer to get a neighbor from the "Input data" warehouse than to compare two vertices (their labels and/or their degrees) stored in its memory. Hence, the number of requests determines the running time of the algorithms. So, we study precisely in this section the worst time complexity and the average time complexity of our three algorithms.

Note that this kind of study is similar to the *query complexity* approach presented in [11]. It has a finer granularity than the complexity analysis in I/O-efficient algorithms or streaming algorithms. Indeed, in the I/O-efficient model (see [16] for a survey), we focus on the number of access disk, while in the streaming model [9], we focus on the number of passes through the data stream.

In Subsect. 4.1, we study the maximum number of requests, by considering for each vertex the worst order in which its neighbors can be retrieved.

In Subsect. 4.2, we study the average number of requests, by considering that for each vertex $u \in V$, its $d(u)$ neighbors can be retrieved in any one of the $d(u)!$ possible orders with a uniform probability. Then, we assume that all the $n!$ labelings of a graph G (in $\mathbb{L}(G)$) can occur with a uniform probability.

Notations. We denote by $d^+(u)$ (resp. $d^-(u)$) the number of right (resp. left) neighbors of u. We denote by $d_{\inf}(u)$ (resp. $d_{\sup}(u)$) the number of neighbors of u having a degree smaller (resp. greater) than that of u. We denote by $\sigma^+(u)$ (resp. $\sigma^-(u)$) the number of right (resp. left) neighbors of u having the same degree.

4.1 The Maximum Number of Requests

In this subsection, we give exact formulas for the maximum number of requests performed by the three algorithms.

Lemma 4.1. *Let $G = (V, L, E)$ be any labeled graph. We denote by $\mathcal{W}\{\mathtt{q}_{\mathcal{A}}(G, L)\}$ (resp. $C_{\mathcal{A}}$) the maximum number of requests made (resp. the cover constructed) by algorithm $\mathcal{A}$ on the labeled*

graph G. One has

$$\mathcal{W}\{\mathtt{q}_{\mathcal{LL}}(G,L)\} = \sum_{u\notin C_{\mathcal{LL}}} d(u) + \sum_{u\in C_{\mathcal{LL}}} (d^-(u)+1) \ , \tag{4}$$

$$\mathcal{W}\{\mathtt{q}_{\mathcal{SLL}}(G,L)\} = \sum_{u\notin C_{\mathcal{SLL}}} d(u) + \sum_{u\in C_{\mathcal{SLL}}} (d_{\sup}(u)+\sigma^-(u)+1) \ , \tag{5}$$

$$\mathcal{W}\{\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(G,L)\} = \sum_{u\notin C_{\overleftarrow{\mathcal{SLL}}}} d(u) + \sum_{u\in C_{\overleftarrow{\mathcal{SLL}}}} (d_{\inf}(u)+\sigma^+(u)+1) \ . \tag{6}$$

We give the proof for $\mathcal{LL}$. Proofs for $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$ are similar.

Proof. Let $G=(V,L,E)$ be a labeled graph. Let $C_{\mathcal{LL}}$ be a cover constructed by $\mathcal{LL}$ on G. Let us consider a vertex u of G. $u\in C_{\mathcal{LL}}$ if and only if it has at least one right neighbor. In the worst case, the processing unit gets all the left neighbors of u before getting a right neighbor. Hence, it makes exactly $d^-(u)+1$ requests to decide that u is in the cover; otherwise, if $u\notin C_{\mathcal{LL}}$, then we have to get all the neighbors of u to decide finally that u is not in the cover (we don't know it has no right neighbor a priori), which generates exactly $d(u)$ requests. The result follows by summing those values for each vertex u of G. □

Theorem 4.1. *Let G be any graph. Let $\mathcal{W}\{\mathtt{q}_{\mathcal{A}}(G)\} = \max_{L\in\mathbb{L}(G)} \mathcal{W}\{\mathtt{q}_{\mathcal{A}}(G,L)\}$ be the maximum number of requests made by algorithm $\mathcal{A}$ on G. One has*

$$\mathcal{W}\{\mathtt{q}_{\mathcal{A}}(G)\} = m + |C_{\mathcal{A}}^{\max}| \ , \tag{7}$$

where $|C_{\mathcal{A}}^{\max}|$ is the maximum size of cover returned by algorithm $\mathcal{A}$ on G.

Proof. We give the proof for $\mathcal{LL}$ and for $\mathcal{SLL}$. The proof for $\overleftarrow{\mathcal{SLL}}$ is similar to the $\mathcal{SLL}$ one.

Proof for $\mathcal{LL}$. Let $G=(V,L,E)$ be any labeled graph and $C_{\mathcal{LL}}$ the cover constructed by $\mathcal{LL}$ on G. We can simplify (4) as follows.

$$\sum_{u\notin C_{\mathcal{LL}}} d(u) + \sum_{u\in C_{\mathcal{LL}}} (d^-(u)+1) = m + |C_{\mathcal{LL}}| \tag{8}$$

since, $\forall u\notin C_{\mathcal{LL}}$, $d(u)=d^-(u)$ and $\sum_{u\in V} d^-(u)=m$. Now, if we maximize (8) by considering all the $n!$ possible labelings of $\mathbb{L}(G)$, we need to maximize the size of $C_{\mathcal{LL}}$. Hence, we obtain $\mathcal{W}\{\mathtt{q}_{\mathcal{LL}}(G)\} = m + |C_{\mathcal{LL}}^{\max}|$.

Proof for $\mathcal{SLL}$. Let $G=(V,L,E)$ be any labeled graph and $C_{\mathcal{SLL}}$ the cover constructed by $\mathcal{SLL}$ on G. We can simplify (5) as follows.

$$\sum_{u\notin C_{\mathcal{SLL}}} d(u) + \sum_{u\in C_{\mathcal{SLL}}} (d_{\sup}(u)+\sigma^-(u)+1) = m + |C_{\mathcal{SLL}}| \tag{9}$$

since, $\forall u\notin C_{\mathcal{SLL}}$, $d(u)=d_{\sup}(u)+\sigma^-(u)$ and $\sum_{u\in V}(d_{\sup}(u)+\sigma^-(u))=m$. Now, if we maximize (9) by considering all the $n!$ possible labelings of $\mathbb{L}(G)$, we need to maximize the size of $C_{\mathcal{SLL}}$. Hence, we obtain $\mathcal{W}\{\mathtt{q}_{\mathcal{SLL}}(G)\} = m + |C_{\mathcal{SLL}}^{\max}|$. □

Corollary 4.1. *The maximum number of requests made by $\mathcal{LL}$ on any graph G (over all its labelings $\mathbb{L}(G)$) having c connected components is*

$$\mathcal{W}\{\mathtt{q}_{\mathcal{LL}}(G)\} \quad = \quad m+n-c \ . \tag{10}$$

Moreover, $\mathcal{W}\{\mathtt{q}_{\mathcal{A}}(G)\} \leq m+n-1$ for $\mathcal{A} = \mathcal{SLL}$ or $\overleftarrow{\mathcal{SLL}}$.

Proof. The results for $\mathcal{LL}$ are derived from Theorem 4.1 and Lemma 3.2. For $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$, note that any cover $C_{\mathcal{A}}$ cannot contain all the vertices of G. □

4.2 The Expected Number of Requests

In this subsection, we give exact formulas expressing the expected number of requests for the three algorithms.

Lemma 4.2. *Let $G = (V, L, E)$ be any labeled graph. We note $\mathbb{E}[\mathtt{q}_{\mathcal{A}}(G, L)]$ (resp. $C_{\mathcal{A}}$) the expected number of requests made (resp. the cover constructed) by algorithm $\mathcal{A}$ on the labeled graph G. One has*

$$\mathbb{E}[\mathtt{q}_{\mathcal{LL}}(G,L)] \quad = \quad \sum_{u \notin C_{\mathcal{LL}}} d(u) + \sum_{u \in C_{\mathcal{LL}}} \frac{d(u)+1}{d^+(u)+1} \ , \tag{11}$$

$$\mathbb{E}[\mathtt{q}_{\mathcal{SLL}}(G,L)] \quad = \quad \sum_{u \notin C_{\mathcal{SLL}}} d(u) + \sum_{u \in C_{\mathcal{SLL}}} \frac{d(u)+1}{d_{\inf}(u)+\sigma^+(u)+1} \ , \tag{12}$$

$$\mathbb{E}[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(G,L)] \quad = \quad \sum_{u \notin C_{\overleftarrow{\mathcal{SLL}}}} d(u) + \sum_{u \in C_{\overleftarrow{\mathcal{SLL}}}} \frac{d(u)+1}{d_{\sup}(u)+\sigma^-(u)+1} \ . \tag{13}$$

We give the proof for $\mathcal{LL}$. Proofs for $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$ are similar.

Proof. Let $G = (V, L, E)$ be any labeled graph. Let $C_{\mathcal{LL}}$ be a cover constructed by $\mathcal{LL}$ on G. Let us consider a vertex u of G. If $u \notin C_{\mathcal{LL}}$, then the algorithm has to get all of its $d(u)$ neighbors; otherwise, it makes $\frac{d(u)+1}{d^+(u)+1}$ requests in expectation before getting one of the $d^+(u)$ right neighbors of u. This value can be explained as follows. If a player is to draw balls from a bag containing a white balls and b black balls until he draws a black ball, not replacing the ball drawn, then the expected number of white balls he will draw is $\frac{a}{b+1}$ (see for example [6]). Now, suppose that the $d(u)$ neighbors of u are balls in a bag, with $d^-(u)$ (resp. $d^+(u)$) white (resp. black) balls. Using $a = d^-(u)$ and $b = d^+(u)$, we obtain $\frac{a}{b+1} + 1 = \frac{d^-(u)}{d^+(u)+1} + 1 = \frac{d(u)+1}{d^+(u)+1}$ requests in average (including the one giving the "black ball"). The result follows by using linearity of expectation. □

Theorem 4.2. *Let G be any graph. Let $\mathbb{E}[\mathtt{q}_{\mathcal{A}}(G)] = \frac{1}{n!}\sum_{L \in \mathbb{L}(G)} \mathbb{E}[\mathtt{q}_{\mathcal{A}}(G, L)]$ be the expected number of requests made by algorithm $\mathcal{A}$ on G, assuming that all the labelings of $\mathbb{L}(G)$ occur with the same probability. For $\mathcal{LL}$ and $\mathcal{SLL}$, we have*

$$\mathbb{E}[\mathtt{q}_{\mathcal{LL}}(G)] \quad = \quad \sum_{u \in V} H(d(u)) \ . \tag{14}$$

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(G)\big] &= \sum_{u\in V}\frac{d(u)+1}{\sigma(u)+1}\big(H(d_{\inf}(u)+\sigma(u)+1)-H(d_{\inf}(u))\big)\\ &\quad-\sum_{u|d_{\inf}(u)=0}\frac{1}{\sigma(u)+1}\ ,\end{aligned}\tag{15}$$

where $H(n)=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ *and* $H(0)=0$.

$\mathbb{E}\big[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(G)\big]$ *is obtained by replacing* $d_{\inf}(u)$ *by* $d_{\sup}(u)$ *in (15).*

Proof. We give the proof for $\mathcal{LL}$ and for $\mathcal{SLL}$. The proof for $\overleftarrow{\mathcal{SLL}}$ is similar to the $\mathcal{SLL}$ one.

Proof for $\mathcal{LL}$. Let $G=(V,E)$ be any graph. We calculate the *contribution* of each vertex $u\in V$ in $\mathbb{E}\big[\mathtt{q}_{\mathcal{LL}}(G)\big]$. Let $L\in\mathbb{L}(G)$ be any labeling on G and $C_{\mathcal{LL}}$ the cover constructed by $\mathcal{LL}$ on the labeled graph $G=(V,L,E)$. Notice that for each vertex $u\in V$, $u\notin C_{\mathcal{LL}}$ if and only if $d^+(u)=0$. Let $\beta_k(u)$ be the proportion of labelings $L\in\mathbb{L}(G)$ for which $d^+(u)=k$. Using (11), the *contribution* of vertex u in $\mathbb{E}\big[\mathtt{q}_{\mathcal{LL}}(G)\big]$ is

$$\beta_0(u)\cdot d(u)+\sum_{k=1}^{d(u)}\beta_k(u)\cdot\frac{d(u)+1}{k+1}\ .\tag{16}$$

Let us compute the value of $\beta_k(u)$. The value of $d^+(u)$ depends only on the label of vertex u compared to those of its neighbors. There are exactly

$$\binom{n}{d(u)+1}\cdot d(u)!\times(n-(d(u)+1))!\tag{17}$$

labelings in which $d^+(u)=k$. Indeed, we assign labels to vertices of G as follows. First, we choose $d(u)+1$ labels among n and u gets the $(k+1)^{\text{th}}$ largest label in order to have $d^+(u)=k$. Then, there remain $d(u)!$ possibilities for labeling neighbors of u and $(n-(d(u)+1))!$ possibilities for the other vertices of G. We obtain $\beta_k(u)$ by dividing (17) by $n!$. Thus, $\forall k\in\{0,\ldots,d(u)\}$, $\beta_k(u)=\frac{1}{d(u)+1}$. Now, we simplify (16) and we get

$$\begin{aligned}\beta_0(u)\cdot d(u)+\sum_{k=1}^{d(u)}\beta_k(u)\cdot\frac{d(u)+1}{k+1} &= \frac{d(u)}{d(u)+1}+\sum_{k=1}^{d(u)}\frac{1}{k+1}\\ &= \frac{d(u)}{d(u)+1}+\sum_{k=2}^{d(u)+1}\frac{1}{k}\ =\ 1+\sum_{k=2}^{d(u)}\frac{1}{k}\ =\ H(d(u))\ .\end{aligned}$$

The result follows by using the linearity of expectation.

Proof for $\mathcal{SLL}$. Let $G=(V,E)$ be any graph. We calculate the *contribution* of each vertex $u\in V$ in $\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(G)\big]$. Let $L\in\mathbb{L}(G)$ be any labeling on G and $C_{\mathcal{SLL}}$ the cover constructed by $\mathcal{SLL}$ on the labeled graph $G=(V,L,E)$. Notice that for each vertex $u\in V$, $u\notin C_{\mathcal{SLL}}$ if and only if $d_{\inf}(u)=0$ and $\sigma^+(u)=0$. Also, note that if $d_{\inf}(u)>0$, whatever the labeling of vertices of G, u is always selected by $\mathcal{SLL}$. Let $\beta'_k(u)$ be the proportion of labelings $L\in\mathbb{L}(G)$ for which $\sigma^+(u)=k$.

1. If $d_{\inf}(u)>0$, whatever the value of $\sigma^+(u)$ (between 0 and $\sigma(u)$ and denoted k by the following), the *contribution* of vertex u in $\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(G)\big]$ is

$$\beta'_k(u)\cdot\frac{d(u)+1}{d_{\inf}(u)+k+1}\ .\tag{18}$$

2. If $d_{\inf}(u) = 0$, then the fact that u is in the cover returned by $\mathcal{SLL}$ or not depends only on label of u compared to those of its neighbors having the same degree. In this case, the *contribution* of vertex u in $\mathbb{E}[\mathtt{q}_{\mathcal{SLL}}(G)]$ is

$$\begin{cases} \beta'_k(u) \cdot \frac{d(u)+1}{k+1} & \text{with } k = 1, \ldots, \sigma(u) \quad \text{if } u \in C_{\mathcal{SLL}}, \\ \beta'_0(u) \cdot d(u) & \text{otherwise.} \end{cases} \tag{19}$$

We obtain the value of $\beta'_k(u)$ by using the same reasoning as for $\mathcal{LL}$. We replace $d(u)$ in $\beta_k(u)$ by $\sigma(u)$ and thus we have, for any vertex u, $\beta'_k(u) = \frac{1}{\sigma(u)+1}$, $\forall k \in \{0, \ldots, \sigma(u)\}$. So, in order to simplify (18), we apply this result for each vertex u such that $d_{\inf}(u) > 0$:

$$\begin{aligned} \sum_{k=0}^{\sigma(u)} \frac{1}{\sigma(u)+1} \cdot \frac{d(u)+1}{d_{\inf}(u)+k+1} &= \frac{d(u)+1}{\sigma(u)+1} \sum_{k=1}^{\sigma(u)+1} \frac{1}{d_{\inf}(u)+k} \\ &= \frac{d(u)+1}{\sigma(u)+1} \big(H(d_{\inf}(u)+\sigma(u)+1) - H(d_{\inf}(u))\big) \ , \end{aligned} \tag{20}$$

and we apply this result on (19), for each vertex u such that $d_{\inf}(u) = 0$:

$$\begin{aligned} \frac{d(u)}{\sigma(u)+1} + \sum_{k=1}^{\sigma(u)} \frac{1}{\sigma(u)+1} \cdot \frac{d(u)+1}{k+1} &= \frac{d(u)}{\sigma(u)+1} + \frac{d(u)+1}{\sigma(u)+1} \left(\sum_{k=1}^{\sigma(u)+1} \frac{1}{k} - 1 \right) \\ &= \frac{-1}{\sigma(u)+1} + \frac{d(u)+1}{\sigma(u)+1} \sum_{k=1}^{\sigma(u)+1} \frac{1}{k} \\ &= \frac{d(u)+1}{\sigma(u)+1} \cdot H(\sigma(u)+1) - \frac{1}{\sigma(u)+1} \ . \end{aligned} \tag{21}$$

The result follows by summing (20) and (21) for each vertex of G. □

Corollary 4.2. *The expected number of requests made by $\mathcal{LL}$, $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$ on a Δ-regular graph is $n \cdot H(\Delta)$, that tends to $n \cdot \log \Delta$ when Δ tends to $+\infty$.*

Proof. As in G for all $u \in V$ we have $d(u) = \Delta$, the result for $\mathcal{LL}$ immediately follows and we also get $d_{\inf}(u) = d_{\sup}(u) = 0$ and $\sigma(u) = d(u)$. Using these values, we can simplify (15) and get the result for $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$. □

Theorem 4.3. *Among $\mathcal{LL}$, $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$, no algorithm can be elected as the best one: there exist graphs for which each algorithm makes an expected number of probes to the instance smaller than the two others.*

Proof. Here, we apply formulas given in Theorem 4.2 to show that, for each algorithm, there exist graphs for which it can be the best in expectation.

- Let S_n be a star with n vertices. If we apply resp. (14), (15) and (15) by replacing $d_{\inf}(u)$ by $d_{\sup}(u)$, for all $n > 2$, we have

$$\mathbb{E}[\mathtt{q}_{\mathcal{LL}}(S_n)] = H(n-1) + n - 1 \ .$$

In a star S_n such that $n > 2$, no vertex has a neighbor having the same degree as it.

For $\mathcal{SLL}$, the $n-1$ leaves of S_n have no neighbor with a smaller degree. The center of S_n has $n-1$ neighbors (the leaves) having a degree smaller than it. Thus, we have

$$\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(S_n)\big] \;=\; (n-1)\cdot 2 + n\big(H(n) - H(n-1)\big) - (n-1) \;=\; n \;.$$

For $\overleftarrow{\mathcal{SLL}}$, the center of S_n has no neighbor having a degree greater than it. Each leaf of S_n has a neighbor (the center) with a greater degree. Thus, we have

$$\mathbb{E}\big[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(S_n)\big] \;=\; n + (n-1)\cdot 2\big(H(2) - H(1)\big) - 1 \;=\; 2n-2 \;.$$

Thus, we can easily see that $\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(S_n)\big] \;<\; \mathbb{E}\big[\mathtt{q}_{\mathcal{LL}}(S_n)\big] \;<\; \mathbb{E}\big[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(S_n)\big]$.

- Let $K_{a,b} = (X \cup Y, E)$ be a complete bipartite graph with $n = a + b$ vertices (where $a = |X|$ and $b = |Y|$). Assuming that $a > b > 4$, we have

$$\mathbb{E}\big[\mathtt{q}_{\mathcal{LL}}(K_{a,b})\big] \;=\; a\cdot H(b) + b\cdot H(a) \;.$$

In a complete bipartite graph $K_{a,b}$ such that $a \neq b$, no vertex has a neighbor having the same degree as it.

For $\mathcal{SLL}$, the a vertices of X have no neighbor with a smaller degree. Each vertex of Y has a vertices (those of X) having a degree smaller than it. thus, we have

$$\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(K_{a,b})\big] \;=\; a(b+1) + b(a+1)\big(H(a+1) - H(a)\big) - a \;=\; ab + b \;.$$

For $\overleftarrow{\mathcal{SLL}}$, the b vertices of Y have no neighbor with a greater degree. Each vertex of X has b vertices (those of Y) having a degree greater than it. thus, we have

$$\mathbb{E}\big[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(K_{a,b})\big] \;=\; b(a+1) + a(b+1)\big(H(b+1) - H(b)\big) - b \;=\; ab + a \;.$$

We can easily see that $\mathcal{SLL}$ is better than $\overleftarrow{\mathcal{SLL}}$ (because $a > b$). We compare $\mathcal{LL}$ with $\mathcal{SLL}$:

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathcal{SLL}}(K_{a,b})\big] - \mathbb{E}\big[\mathtt{q}_{\mathcal{LL}}(K_{a,b})\big] \;&=\; ab + b - a\cdot H(b) - b\cdot H(a) \\ &=\; \frac{ab}{2} - a\cdot H(b) + \frac{ab}{2} - b\cdot H(a) + b \;>\; 0\end{aligned}$$

because $\frac{b}{2} > H(b)$ when $b > 4$ and $\frac{a}{2} > H(a)$ when $a > 4$.

Therefore, when $a > b > 4$, $\mathcal{LL}$ produces, in expectation, a smaller number of requests than $\mathcal{SLL}$ and $\overleftarrow{\mathcal{SLL}}$.

- Let $CK_{l,w}$ be a *necklace* with $n = l \times w$ vertices. A necklace $CK_{l,w}$ is a cycle of l complete graphs where each complete graph K_i ($i \in \{1, \ldots, l\}$) has w vertices: $w-2$ vertices of degree $w-1$ and 2 distinct vertices a_i and b_i of degree w, called *connectors*, which connect K_i to its previous and to its following neighbors in the cycle (see Fig. 4 for an illustration).

Assuming that $l > 1$ and $w > 4$, we have

$$\mathbb{E}[\mathtt{q}_{\mathcal{LL}}(CK_{l,w})] \;=\; l(w-2)\cdot H(w-1) + 2l\cdot H(w) \;=\; n\cdot H(w-1) + \frac{2l}{w}\ .$$

In a necklace $CL_{l,w}$, each vertex belonging to the l complete graphs except the $2l$ connectors a_i and b_i has $w-3$ neighbors having the same degree as it. Each connector has 2 neighbors having the same degree as it.

For $\mathcal{SLL}$, the $l(w-2)$ different vertices of the connectors have no neighbor with a smaller degree. Each connector has $w-2$ neighbors (the vertices of the complete graph it belongs) having a degree smaller than it. Thus, we have

$$\begin{aligned}\mathbb{E}[\mathtt{q}_{\mathcal{SLL}}(CK_{l,w})] &= l(w-2)\cdot\frac{w}{w-2}\cdot H(w-2) + 2l\cdot\frac{w+1}{3}(H(w+1)-H(w-2)) - \frac{l(w-2)}{w-2}\\ &= n\cdot H(w-2) + \frac{2l}{3}\cdot\frac{3w^2-1}{w(w-1)} - l\ .\end{aligned}$$

For $\overleftarrow{\mathcal{SLL}}$, the $2l$ connectors have no neighbor with a greater degree. Each vertex (except the connectors) has 2 neighbors (the connectors of the complete graph it belongs) having a degree greater than it. Thus, we have

$$\begin{aligned}\mathbb{E}[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(CK_{l,w})] &= 2l\cdot\frac{w+1}{3}\cdot H(3) + l(w-2)\cdot\frac{w}{w-2}(H(w)-H(2)) - \frac{2l}{3}\\ &= n\cdot H(w) - \frac{5n}{18} + \frac{5l}{9}\ .\end{aligned}$$

We compare them:

$$\begin{aligned}\mathbb{E}[\mathtt{q}_{\mathcal{LL}}(CK_{l,w})] - \mathbb{E}[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(CK_{l,w})] &= \frac{5n}{18} + \frac{2l}{w} - \frac{14l}{9}\\ &= \frac{l(5w^2-28w+36)}{18w} \;>\; 0\end{aligned}$$

when $w > 3$, because the polynomial $5w^2 - 28w + 36$ has two roots: 2 and 3.6; and is positive $\forall w \le 2$ and $\forall w \ge 3.6$.

$$\begin{aligned}\mathbb{E}[\mathtt{q}_{\mathcal{SLL}}(CK_{l,w})] - \mathbb{E}[\mathtt{q}_{\overleftarrow{\mathcal{SLL}}}(CK_{l,w})] &= \frac{5n}{18} + \frac{n}{w-1} - \frac{2l}{3w(w-1)} - \frac{23l}{9}\\ &= \frac{l(5w^3-33w^2+46w-12)}{18w(w-1)} \;>\; 0\end{aligned}$$

when $w > 4$, because the polynomial $5w^3 - 33w^2 + 46w - 12$ has three roots: 0.34, 1.48 and 4.78; and is positive $\forall w \in [0.34, 1.48]$ and $\forall w \ge 4.78$.

Hence, when $l > 1$ and $w > 4$, $\overleftarrow{\mathcal{SLL}}$ produces, in expectation, a number of requests smaller than $\mathcal{LL}$ and $\mathcal{SLL}$.

□

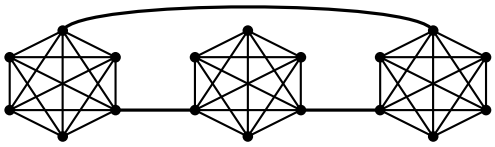

Figure 4: Example of a necklace with $l = 3$ and $w = 6$

5 Conclusion

We have presented and analyzed three algorithms for the VERTEX COVER problem, which are suitable to the severe constraints of our model: they don't need to modify the input graph, they don't need a large memory on the processing unit, and they don't need to read and/or modify the solution computed during the execution. They are adapted to the construction of a vertex cover in huge graphs on a basic computer.

If the degrees of the vertices are directly available, the three algorithms can be used; otherwise, only $\mathcal{LL}$ can be applied. To compare these three methods, in Sect. 3, we have given exact (analytical) formulas for the expected size of the cover returned by these algorithms (we also proved that for any graph there exist labelings for which $\mathcal{LL}$ give the optimal cover). We proved that, based on this measure, no algorithm among these three can be elected as the best one for all graphs.

To go further in the analysis, in Sect. 4, we have given exact formulas expressing the maximum and the expected number of requests made by the three algorithms (to the system containing the input data) to construct the solution. Again, based on this running time complexity measure, we have proved that none of the three algorithms can be elected has the best (i.e. fastest) one.

All our analytical formulas can help a user to choose among our three algorithms based on potential knowledges on the input graph (that may be given by the domain of application). They can be used to balance between precision and complexity.

We can also remark that the three algorithms can easily be executed in parallel if each processing unit manages a subset (not necessarily consecutive) of vertices.

We believe that $\mathcal{SLL}$ is the algorithm constructing the smallest vertex cover in average for "almost all" graphs[3] if the degrees are available. A perspective is to prove that; this is probably hard for all graphs; some experiments results could also be helpful.

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley, New York, 1992.

[2] E. Angel, R. Campigotto, and C. Laforest. Algorithms for the Vertex Cover Problem on Large Graphs. Technical Report No 1, IBISC – Université d'Evry, 2010.

[3] This opinion is also based on analysis on other class of graphs and several experimentations not included in this paper.

[3] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer, 1999.

[4] D. Avis and T. Imamura. A List Heuristic for Vertex Cover. *Operations Research Letters*, 35:201–204, 2006.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., New York, 1979.

[6] M. Glaymann. Où le Premier n'est pas Toujours Premier... *Educational Studies in Mathematics*, 7(1-2):83–88, July 1976.

[7] G. Karakostas. A Better Approximation Ratio for the Vertex Cover Problem. *ACM Transactions on Algorithms (TALG)*, 5(4), October 2009.

[8] R. Lippert, R. Schwartz, G. Lancia, and S. Istrail. Algorithmic Strategies for the Single Nucleotide Polymorphism Haplotype Assembly Problem. *Brefings in Bioinformatics*, 3(1):23–31, March 2002.

[9] T. C. O'Connell. *Fundamental Problems in Computing*, chapter A Survey of Graph Algorithms under Extended Streaming Models of Computation, pages 455–476. Springer Science + Business Media, 2009.

[10] C. H. Papadimitriou and M. Yannakakis. Optimization, Approximation and Complexity Classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.

[11] M. Parnas and D. Ron. Approximating the Minimum Vertex Cover in Sublinear Time and a Connection to Distributed Algorithms. *Theoretical Computer Science*, 381:183–196, Avril 2007.

[12] S. Pirzada and A. Dharwadker. Applications of Graph Theory. *Journal of The Korean Society for Industrial and Applied Mathematics (KSIAM)*, 11(4):19–38, 2007.

[13] C. Savage. Depth-First Search and the Vertex Cover Problem. *Information Processing Letters*, 14(5):233–235, July 1982.

[14] K. Smith. Genetic Polymorphism and SNPs. Available on `http://www.cs.mcgill.ca/~kaleigh/compbio/snp/snp_summary.html`, February 2002.

[15] U. Stege. *Resolving Conflicts in Problems from Computational Biology*. PhD thesis, ETH Zurich, Institute of Scientific Computing, 2000.

[16] J. S. Vitter. *Algorithms and Data Structures for External Memory*, volume 2. Foundations and Trends in Theoretical Computer Science, Boston – Delft, 2009.

[17] Y. Zhang, Q. Ge, R. Fleisher, T. Jiang, and H. Zhu. Approximating the Minimum Weight Weak Vertex Cover. *Theoretical Computer Science*, 363(1):99–105, 2006.